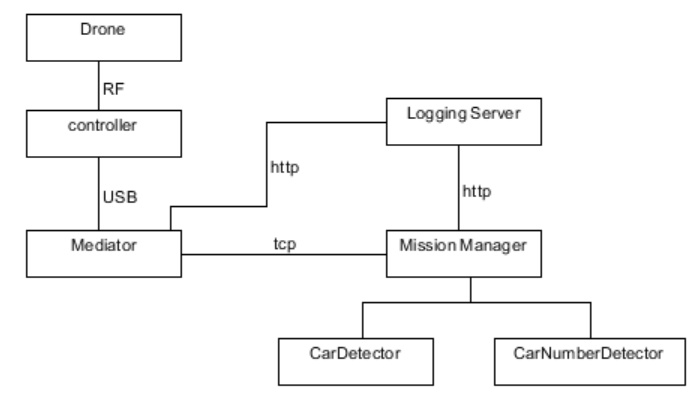
**Parking Drone**



**DEVELOPER GUIDE**

הקדמה



# תיאור קצר של תתי המערכות השונות:

* **:Drone**

רחפן מסוג DJI MATRICE 210 אשר נרכש במיוחד בשביל הפרויקט, במסגרת הפרויקט לא בוצעו הרחבות לחבילת התוכנה הבסיסית שמגיעה עם הרחפן.

* **Controller:**

השלט אשר מגיע עם הרחפן, שלט המסוגל לדבר עם הרחפן באמצעות אותות RF הנשלחות ממנו אל הרחפן.

* **Mediator:**

אפליקציית אנדרואיד השולטת בפעולת הרחפן. מחוברת באמצעות USB TYPE C אל השלט. האפליקציה מקבלת פקודות משרת מנהל המשימות ומעבירה אותם אל הרחפן באמצעות השלט.

* **Logging server:**

שרת לוגים אשר נמצא תחת האתר <https://floating-fjord-95063.herokuapp.com/log>.

משמש לשמירת הלוגים של הרחפן בצורה הנוחה וקלה לצפייה ולניתוח.

* **Mission manager server:**

התוכנה שמנהלת את כל השליטה על הרחפן. ואת התקשורת עם תתי המערכות השונות. תפקידה ביצוע משימות וניהולם.

* **Car detector:**

מודול תוכנה שממש אלגוריתם אשר מקבל תמונה ומחזיר איכן נמצאים בתמונה רכבים, ממומש בC++ ועטוף ב-Python לצורכי נוחות.

* **Car plate detector:**

מודול תוכנה אשר מקבל תמונה של רכב ומחזיר string המכיל את מספר הרכב משתמש בשירות של AWS (Amazon).

* **Simulator:**

כיוון שלא תמיד ניתן להטיס את הרחפן, פיתחנו Simulator לרחפן שניתן בעזרתו לדמות את ההטסה של הרחפן ולבדוק את המודולים השונים במערכת.

תיאור המערכת

Mediator

הmediator הינו רכיב תוכנה אשר מותקן על גבי טלפון סלולרי. הטלפון הסלולרי מחובר בחיבור type c אל שלט הרחפן. השלט והmediator מתקשרים באמצעות dji sdk. כיוון שהapi של השלט מסובך ואינו נוח למפתחים, בחרנו לעטוף את הapi הזה באמצעות רכיב תוכנה אשר ניקרא mediator. ה mediator חושף api פשוט ונוח לניהול הרחפן(ניתן לראות פירוט של פרוטוקול זה במסמך ADD).

על מנת להריץ את הפרויקט הmediator יש לפתוח אותו באמצעות android studio.

לינק לדוקומנטציה של DJI SDK <https://developer.dji.com/api-reference/android-api/Components/SDKManager/DJISDKManager.html>

נסקור את מודולים השונים:

**Loading.java** – למחלקה זו שתי פעולות עיקריות

1. ביצוע רגיסטרציה מול DJI ללא רגיסטרציה לא נוכל לגשת ל SDK במלואו ולא נוכל לשלוח פקודות לרחפן במידה ושינתם את שם האפליקציה עליכם לבצע רישום מחדש של האפליקציה באתר המפתחים של DJI.
2. בקשת הרשאות לאפליקציה (כמו הרשאת כתיבה למערכת הקבצים)

**MainActivity.java** – למחלקה זו שלוש פעולות עיקריות

1. ניהול הדף שנראה בעת טיסת הרחפן
2. אתחול השכבה הלוגית
3. פענוח stream המתקבל ממצלמת הרחפן ושמירתו כתמונה במערכת הקבצים של מכשיר האנדרואיד.

**BLManager.java** –

תפקידה להכין את מערכת הקבצים לקראת שמירת תמונות מהרחפן ולאתחל את רכיב ההאזנה SocketManager.java, ולבצע אתחול, ווידו תקשורת עם הרחפן.

**SocketManager.java** –

מחלקה שמטרתה פירסור ההודעות המגיעות ממנהל המשימות והעברת הטיפול בהם למחלקה TaskManager.java.

Decoder.java –

מחלקה שבאמצעותה ה SocketManager מפרסר את המשימות לאחד מאובייקטי המשימות. הפרוטוקול הנתמך מתואר במסמך התיכון של המערכת.

**TaskManager.java** –

למחלקה ארבע טרדים המטפלים בסוגי המשימות העשויות להתבצע במקביל בקשות המיקום והמצב שהנשלחות מתקבלות באופן תכוף ולכן לרוב מתבצעות במקביל לבקשת משימה כמו כן בקשת עצירה תפקידה לעצור את המשימה הנוכחית ולכן גם היא עשויה להתקבל במקביל לשלושת האחרות.

**Missions package** –

ניתן למצוא את המשימות בהם תומכת אפליקצית האנדרואיד ופרוטוקול התקשרות בין מנהל המשימות לבין האפליקציה.

נשים לב כי הלוגיקה של כל משימה נקראת מהמתודה start אך אם נתבונן במחלקות השונות נבחין כי בחלקם הלוגיקה כולה ממומשת בנקודה זו וביתר הלוגיקה מואצלת לאחת המחלקות תחת הנתיב app/java/BL/Drone/M210Manager למה זה ככה?

בשלב מאוחר יותר של פיתוח הפרויקט רצינו לתמוך במספר סוגי רחפנים, לכל רחפן קריאות מעט שונות לך DJI SDK ולכן התבקש לייצר DroneFactory שיחזיר את הרכיב שיבצע את הלוגיקה בהתאם לסוג הרחפן. אך משקולי זמן תהליך זה נעצר.

לבסוף רוב פיתוח התקשורת עם הרחפן נימצא ב M210Manager.

**תמיכה במשימה חדשה:**

על מנת לתמוך במשימה חדשה נבצע את השלבים הבאים:

1. נוסיף ל switch case שבמחלקה Decoder.java את ההודעה שתגרום ליצרתה.
2. ניצור משימה ונירש את המחלקה האבסטרקטית Mission
3. נממש את הלוגיקה תחת הפונקציה start
4. נממש את הפונקציה encode המהווה את ערך ההחזרה של המשימה וההודעה כי היא הסתיימה למנהל המשימות (תוכנת המחשב)
5. במידה ויש פעולות חריגות שנרצה לבצע אם המשימה נעצרה באמצע ביצועה נממש את הפונקציה stop

באופן שונה נוכל לתמוך ברחפן חדש ומשימות חדשות ע"י מימוש המנשק IDrone ושינוי ה Instance המוחזר ב DroneFactory ( ניתן לשנות את DroneFactory כך שבאמת יממש את תבנית העיצוב שהוא קרוי על שמה). לאחר מכן נעבור על המשימות ונשנה את הקריאות בפונקציה start לקריאות כדוגמת DroneFactory.getDroneManager(MyDrone.class).missionName

**קונפיגורציה**

תחת SharedClasses

DST\_ADDRESS – שנו את הקונפיגורציה להיות ה IP של המחשב עליו רץ מנהל המשימות.

DST\_PORT – שנו את הקונפיגורציה להיות ה PORT שבו מקשיב התהליך בו רץ מנהל המשימות.

Logger.java – מחלקה זו שולחת את הלוגים שנכתבים באת ריצת האפליקציה לשרת הלוגים, השרת שבשימוש כרגע הוא שרת שלנו ולכן תצרכו להחליף את כתובת לשרת חדש שעליו תריצו את שרת ה logging. השורה אותה תצרכו להחליף היא: בפונקציה sendGetRequest שנו את הכתובת במשתנה Url לכתובת של השרת שלכם.

Node Logging Server

האפליקציה כתובה ב nodejs ומשתמשת ב framework – express.

מבנה האפליקציה:

App.js – הקובץ מנווט את כלל בקשות ה http ליעדם.

Routes/create – מטפל בבקשות get שנשלחו לנתיב create

Routes/log – מטפל בבקשות get ו post שנשלחו לנתיב log

Routes/empty – מטפל בבקשות get שנשלחו לנתיב empty

שלבים להרצה:

1. העתיקו את תקיית שרת הלוגים למחשב הרצוי והתקינו node js. וודאו כי קיים משתנה סביבה עבור npm לאחר ההתקנה.
2. הריצו את הפקודה npm install הפקודה אמורה להתקין את כלל הספריות הדרושות להרצת הקוד.
3. הריצו את הפקודה npm start
4. פתחו את הדפדפן וגילשו לכתובת <http://yourUrl/create> יוסבר ב API

כדי לשנות את הפורט ומאפיינים נוספים ערכו את הקובץ bin/www.

בעיה מרכזית – מערכת הקבצים בה תומך הפרויקט היא של שרתים של החברה herukuapp.com ומשתמש בבסיס נתונים מסוג postgres כדי להפעיל את המערכת על כל מחשב אחר תצרכו לשנות את המודול הכותב למערכת את הלוגים.

תחת הנתיב public/routes שנו את הכתיבה לבסיס הנתונים שלכם. כאמור עליכם לשנות כעת את הכתובת ב Logger.java.

API

<http://yourUrl/create> GETיצור את ה database בו יכתבו הלוגים.

<http://yourUrl/empty> GETמנקה את הלוגים שנכתבו עד כה.

<http://yourUrl/log> GETמחזיר את כל הלוגים שנכתבו עד כה.

<http://yourUrl/log> POST רושם batch של לוגים שהתקבלו למערכת הקבצים.

Mission manager (server)

בתוך ה-solution בשם DroneServer יש מספר פרויקטים כאשר הפרויקט בשם DroneServer הוא הפרויקט המהווה את מנהל המשימות ויתר הפרויקטים הינם פרויקטי בדיקות ברמות השונות וUI.

הפרוייקט בשם DroneServer מחולק לשלוש שכבות.

## Data Layer

בתיקיה זו נמצאים:

* מסד נתונים DroneDB
* קובץ DB.cs המשמש לתקשורת עם מסד הנתונים ומכיל את השאליתות הרלוונטיות.

במסד הנתונים ישנם שלוש טבלאות

טבלת ה BorderPoint מכילה נקודות GPS המשמשות לתיחום חניה כאשר כל נקודה מקושרת לחניה אותה היא תוחמת.

טבלת ה Parking מכילה את פרטי החניה.

טבלת רכבים מורשים המכילה את מספרים הרכבים המורשים לחנות בחניה.

### Presentation Layer

שיכבה זו אחראית על הגוי למפתחים.

בתיקיה זו נמצאים:

* קובץ GUI.cs של VS (windows form file), נדגיש כי זה אינו ה-GUI החשוף למשתמש אלא GUI לפיתוח בעל פונקציונליות נוספת.
* תיקיית observers קיימות מחלקות הממשות Observer pattern על מנת להציג מידע משתנה בשיטת mvc

## Shared Classes

בתיקיה זו נמצאים מחלקות כלליות אשר נמצאות בשימוש אצל כלל רכיבי הפרויקט.

## Business Layer

בתיקיה זו נמצאים:

* **תיקיית CV** בה נמצאים רכיבי עיבוד התמונה של הפרוייקט. רכיבים חדשים יכתבו מהיסוד ויתווספו לתיקיה זו. נדון בהרחבה על רכיבים אלו בהמשך.
* **תיקיית Missions**:

בתיקיה זו נמצאים אובייקטים המהוים משימה או תת-משימה בשרשרת המשימות של הרחפן.

כל משימה ממשת לוגיקה מסויימת, לעיתים משפיעה על רכיבים פנימיים ולעיתים על הרחפן עצמו.

מחלקת האב של כל המשימות הינה Mission הכוללת את המתודות הבאות:

public abstract void execute();

public virtual void done(Response response)

public void register\_to\_notification(notify\_finished func)

כדי ליצור משימה נירש את המחלקה האבסטרקטית Mission.

אנו מוכרחים למשש את הפונקציה execute בה נכתוב את כל הלוגיקה של אותה משימה.

עם סיום המשימה עלינו לקרוא לפונקציה done מאחר ולוגיקה נוספת תלויה בסיום ביצוע המשימה.

נוכל להגדיר פונקציות שיופעלו בסיום המשימה באמצעות הפונקציה register\_to\_notification .

המחלקות הבסיסיות היורשות מ-Mission הינן complexMission LeafMission

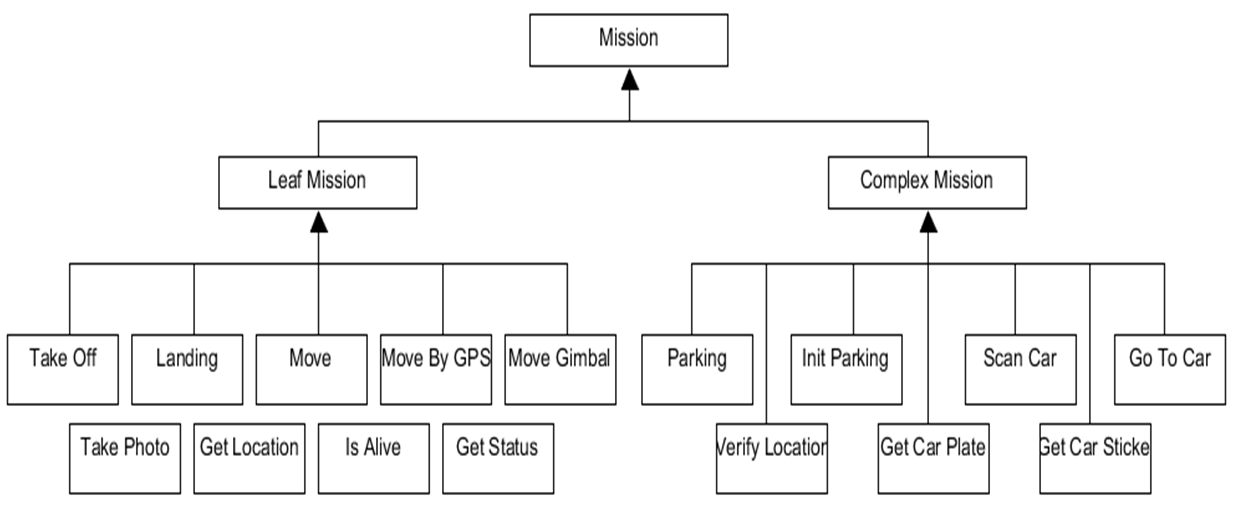
LeafMission הינה משימה המבצעת לוגיקה אשר אינה תלויה בביצוע של משימה אחרת, ולאחר מכן מתריעה למשימה המורכבת שקראה לה שהיא נגמרה.

complexMission הינה משימה שמורכבת מאחת או יותר תתי משימות (מורכבות או פשוטות) אותן היא מבצעת לפי סדר הכנסתן לרשימה, בסיומה כמו כל משימה היא מתריעה למי שהפעיל אותה.

\*לצורך הרחבה של פונקציונליות ההתראה כל משימה תממש את הפונקציה:

public override void notify(Response response)

מבנה היררכי של המחלקות:



באמצעות מבנה זה אנו מאפשרים הרחבה של הקוד באופן קל ומודלורי.

* **BLManager.cs** הינה המחלקה המקשרת בין כל רכיבי הפרויקט ובפרט מפעילה את המשימה הראשונה והכללית ביותר של סריקת החניה:

public MissionWraper startMission(Parking parking)

{

num\_of\_scaned\_cars = 0;

LocationManager.init();

ParkingMission start\_mission = new ParkingMission(parking);

return new MissionWraper(start\_mission);

}

על מנת להוסיף/לשכתב את הלוגיקה הקיימת יש ליצור משימה חדשה (במקום start\_mission) ולתת אותה כפרמטר לwrapper או לעקוב פנימה אחר הלוגיקה של משימת החניה ולערוך/להוסיף במיקום הרצוי את הלוגיקה המתווספת.

CarDetector.cs

מחלקה זו עושה שימוש ב YOLO V3 neural network, המחלקה פותחת תהליך בפייתון המריץ את Yolo V3 עם פרמטרים שהותאמו לזיהוי רכבים מתמונה עלית מגובה רב הבאים לידי ביטוי בקובץ המשקלים הנוצר מאימון מקדים של הרשת על מאגר מידע. אם תרצו לשפר את ביצועי הרשת תאלצו לאמן אותה מחדש ולהחילף את קובץ המשקלים שלכם.

אם נפתח את הקובץ car\_detector.py נוכל לראות את הפרמטרים הדרושים להרצת הרשת כאשר האחרון שבהם הינו הנתיב לתמונה אותה נרצה לנתח.

args=['darknet\_no\_gpu.exe','detect','yolov3-aerial.cfg','yolov3-aerial.weights','cfg/coco.data', os.path.basename(base\_photo\_name)]

כדי להריץ את הרשת באופן ידני תחת הנתיב C:\Users\barrh\Desktop\pk\ParkingDrone\car\_detector\build\darknet\x64

הריצו את הפקודה :

Darknet\_no\_gpu.exe detect yolov3-aerial.cfg yolov3.aerial.weights cfg/coco.data IMAGE\_PATH

CarPlateDetector.cs

מחלקה זו מחלצת מספר רכב מתמונה ועושה שימוש ב amazon rekognition service.

באופן דומה גם כן המחלקה פותחת תהליך פייתון המשתמש ב boto3 client המשמש לתקשורת עם amazon s3 buckets.

המחלקה יוצרת חיבור עם s3 bucket מעלה תמונה לבקט ומריצה rekognition על התמונה שהועלתה. הפלט שחוזר מכיל את כל המספרים שזוהו.

חשוב מאוד: יש צורך להירשם לשירות זה, מאחר והוא בתשלום, לינק למדריך: <https://docs.aws.amazon.com/rekognition/latest/dg/getting-started.html> .

# מערכות נוספות שחשוב להכיר

## VerifyLocationBySift.class , VerifyLocationByTemplateMatching.class

אלגוריתמים אלו לאימות מיקום מתבססים על שתי תמונות כשאר תמונה א מוכלת בתמונה ב, האלוגריתם מאתר את המיקום של תמונה א בתוך תמונה ב. כאשר תמונה א במקרה שלנו היא תצלום כלפי מטה של הרחפן ברגע נתון ותמונה ב היא התמונה העלית הראשונה של החניה.

באופן זהה בכדי להריץ את כל אחד מהאלגורתמים אנו פותחים תהליך בפייתון אך הסטייות המתקבלות הינם עצומות מאחר בתנאים שאינם תנאי מעבדה התמונות עבורות עיותים קיצוניים המקשים על פעילות האלגוריתמים ולכן הם אינם בשימוש.

## Simulator

קיים סימולטור של DJI, אשר ניקרא DJI Assistant 2 for Matrice אשר יכול לדמות טיסה של רחפן אך אינו מדמה צילום עם מצלמה.

כדי שנוכל לבדוק את הקוד עם המצלמה בתנאי מעבדה (ללא צורך ברחפן) פותח סימולטר.

כדי לסמלץ טיסה אמיתית הסימולטור מחזיר ערכים רנדומלים ביחס לסטייה המקונפגת בקובץ הקונפיגורציה.

על מנת להריץ את הסימולטור תחת נתיב drone\_mock/ הריצו את הפקודה python BL.py זאת רק לאחר שהרמתם את התוכנה של מנהל המשימות וקינפגתם את הסימולטור לנוחיותכם.

**קונפיגורציה:**

פתחו את הקובץ BaseConfig.json

קישוריות - במידה ושינתם את הפורטים של מנהל המשימות אנא קנפגו מחדש תחת camera ו transport , כמו כן אם אתם מריצים את הסימולטור על מחשב אחר שנו את ip\_of\_server תחת transport לכתובת הרלוונטית.

בחירת מצלמה – אם נחבר עבור השדה camera\_type ב SimpleCamera אז בכל פעם שהרחפן יצלם תמונה התמונה שתתקבל תהייה ע"פ הערך המספרי של שמה ותילקח מהתקייה images כאשר התמונה בשם 1.jpg תבחר לפני התמונה בשם 2.jpg. אם נבחר ב AerialViewCamera התמונה הראשונה שתצלום תתפס כתמונת בסיס של מגרש החניה ויתר התמונות יגזרו באמצעות המיקום שיחסי של הרחפן ברגע הצילום לעומת התמונה המקורית.

ישנם פרמטרים נוספים הקשורים למהירות הרחפן, סטייה אפשרית, וקונפיגורציות העוזרות לתרגם פיקסלים למטרים ולנקודות gps המותאמים למצלמה הספציפית שבה השתמשנו.